QuickTime Streaming Guide

2006-01-10

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to QuickTime Streaming Guide

QuickTime streaming allows QuickTime movies to play on a client computer while being transmitted from a server. This document describes the basics of QuickTime streaming and the types of protocols supported.

The document also describes in detail the requirements for packetizer components that divide movies into packets for streaming and reassemble packets into movies for display on the receiving end. It lists the functions relevant to media packetizers and reassemblers, along with common streaming error codes.

Note: This document was previously titled *QuickTime Streaming*.

You need to read this chapter if you want to do any of the following:

- play streamed movies within your application
- write RTP server software that transmits streamed QuickTime movies
- write a media packetizer
- write a media reassembler

Organization of This Document

This book is divided into the following chapters:

- "About QuickTime Streaming" (page 9) discusses the basics of QuickTime streaming and the variety of protocols that can be used to stream QuickTime movies.
- "Using QuickTime Streaming" (page 17) describes from a programming perspective how to create, send, and receive streamed movies.
- "Media Packetizers" (page 31) describes the media packetizer and packet builder, and tells you how to build a packetizer.
- "Packet Reassemblers" (page 45) tells you how to build a packet reassembler.

See Also

The following Apple books cover related aspects of QuickTime programming:

- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.
- *QuickTime Movie Creation Guide* describes some of the different ways your application can create a new QuickTime movie.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

About QuickTime Streaming

This chapter discusses the basics of QuickTime streaming and the variety of protocols that can be used to stream QuickTime movies.

Streaming involves sending movies from a **server** to a **client** over a network such as the Internet. The server breaks the movie into **packets** that can be sent over the network. At the receiving end, the packets are reassembled by the client and the movie is played as it comes in. A series of related packets is called a **stream.**

QuickTime Streaming extends the QuickTime software architecture to support the creation, transmission, and reception of multimedia streams. This allows QuickTime programmers to create applications that receive multimedia in real time, and to create authoring and editing tools that work with streaming content. Existing applications that play QuickTime movies can play real-time streaming movies with little or no code change.

Streaming is different from simple file transfer, in that the client plays the movie as it comes in over the network, rather than waiting for the entire movie to download before it can be played. In fact, a streaming client may never actually download a streaming movie; it may simply play the movie's packets as they come in, then discard them.

QuickTime movies can be streamed using a variety of protocols, including

- HTTP (Hyper Text Transport Protocol)
- FTP (File Transfer Protocol)
- RTP (Realtime Transport Protocol).

HTTP and FTP are essentially file transfer protocols. Any QuickTime movie saved using the QuickStart option can be streamed using these protocols because the QuickTime client software is able to start playing the movie before the entire file has arrived.

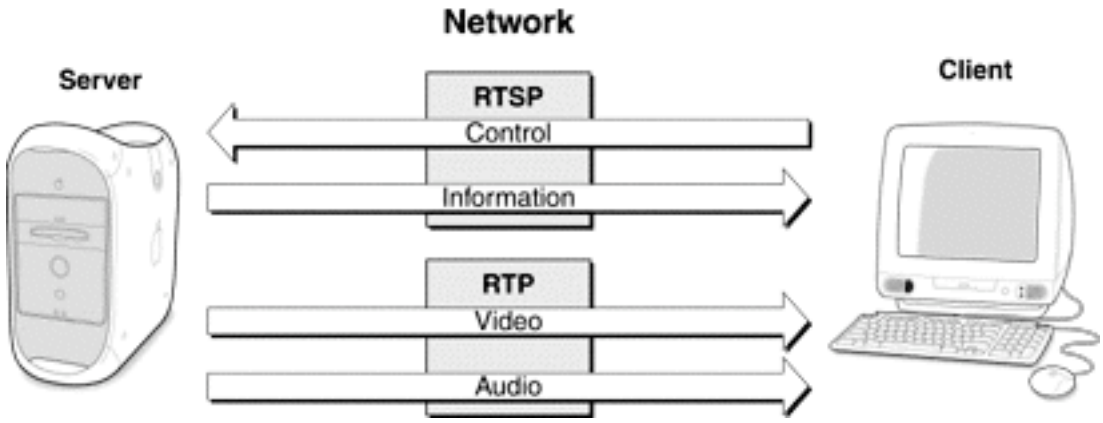
RTP is used for **real time streaming.** The movie packets are sent in real time, so that a one-minute movie is sent over the network in one minute. The packets are time-stamped, so they can be displayed in time-synchronized order. Because packets are sent in real time, RTP streaming works with **live content** in addition to previously-recorded movies. It can even carry a mixture of the two.

Real-time streams can be sent one-to-one (**unicast)** or one-to-many (**multicast).**

Unicast Streaming

In a unicast, the client contacts the server to request a movie using **RTSP** (Real Time Streaming Protocol). The server then replies to the client over RTSP with information describing the movie as a **streaming session.** A streaming session consists of one or more streams of data, such as a video stream and an audio stream. The server tells the client how many streams to expect and gives details on each stream, such as the media type and codec. The actual streams are then sent to the client over RTP. When a QuickTime movie is streamed over RTP, each track in the movie is sent as a separate stream as shown in Figure 1-1.

Figure 1-1 Unicast streaming using RTSP



A stream can contain live content, such as a stock ticker or a radio broadcast, or stored content, such as a video track from a QuickTime movie. When a client is receiving unicast streams from stored content, the client's movie controller includes a "thumb" that allows the user to jump to any point in the movie. This gives the client random access to long movies without having to download an entire movie or store it locally. The client simply asks the server to begin streaming the movie from a new point (see Figure 1-2).

Figure 1-2 Movie controller with thumb

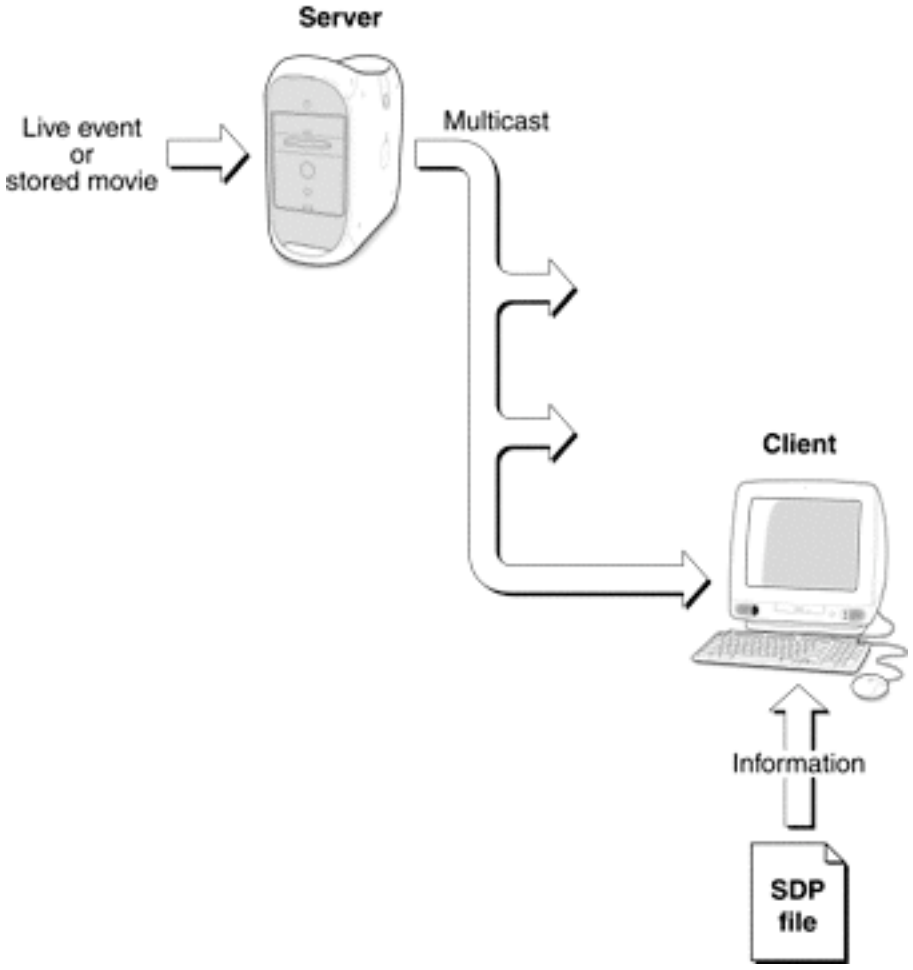
Note: RSTP uses TCP/IP transport, but RTP uses low-level UDP/IP transport. If the client is behind a firewall that doesn't pass UDP, the movie will set up without error, but no media will be displayed. RTSP Proxy servers for several operating systems are available from Apple to help resolve firewall problems.

Multicast Streaming

In a multicast, one copy of each stream is sent over each branch of a network. This reduces the amount of network traffic required to send the streams to large numbers of clients. A client receives the streams by "joining" the multicast.

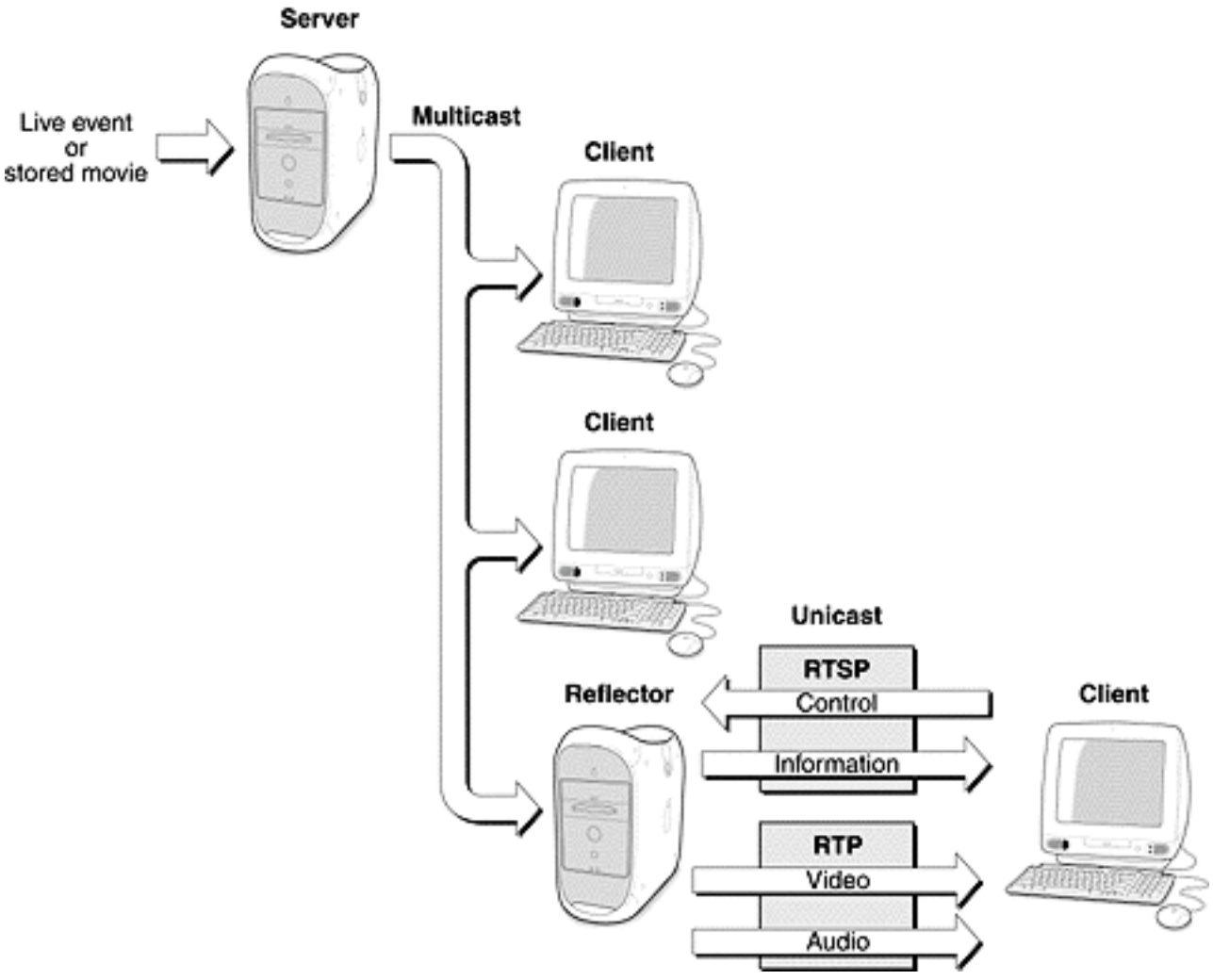
The client finds out how to join the multicast by opening an **SDP** (Session Description Protocol) file. The SDP file contains the information needed to join the multicast, such as group address and port number, as well as the stream description information that would come over RTSP for a unicast. See Figure 1-3. SDP files are commonly posted on web servers to announce upcoming multicasts.

Figure 1-3 QuickTime lets your computer join a multicast by opening an SDP file.



Not all routers support multicasting. QuickTime clients behind routers that don't implement multicasting can still receive a multicast by requesting the streams from a **reflector.** A reflector is an RTSP server that joins a multicast, then converts the multicast into a series of unicasts, passing the streams to clients who request them (see Figure 1-4). The original server may be sending live content, such as a concert or a news broadcast, or a previously-recorded movie. The reflector is always sending "live" data, passing the streams in real time.

Figure 1-4 You can also receive a multicast through a reflector



When a QuickTime client is viewing a multicast or a live unicast, the user's movie controller has no "thumb" control (see Figure 1-5). The user can stop or resume the display, and may have audio controls if the movie includes a sound stream, but there is no way to skip forward or back in a live transmission or a multicast.

Figure 1-5 Movie controller without thumb

Streaming QuickTime Over RTP or HTTP

QuickTime supports streaming over RTP and HTTP. The main advantages of streaming over RTP are:

- RTP can be used for live transmission and multicast.
- Real-time streaming allows the user to view long movies or continuous transmissions without having to store more than a few seconds of data locally.
- Using RTP transmission under RTSP control, a user can skip to any point in a movie on a server without downloading the intervening material.
- You can stream a single track over RTP, whereas HTTP streams only whole movies. RTP streams can be incorporated in a movie using **streaming tracks.** A streaming track is a track in a QuickTime movie that contains the URL of the streaming content.
- A QuickTime movie that contains streaming tracks can also include non-streaming tracks whose media exist on the client's computer. This allows a live transmission, or data stored on the Internet, to be incorporated into a movie along with material stored on the client's hard drive or distributed over CD-ROM.
- RTP uses UDP/IP protocol, which doesn't attempt to retransmit lost packets. This allows multicasts as well as live streams, both cases where retransmission would not be practical.

The main advantages of streaming over HTTP are:

- HTTP uses TCP/IP protocol to ensure that all movie packets are delivered, retransmitting if necessary.
- HTTP does not attempt to stream in real time. To stream in real time, the bandwidth of the network must be greater than the data rate of the movie. If there is not enough bandwidth to transmit the movie in real time, streaming by HTTP allows the client to store the data locally and play the movie after enough has arrived.
- Most firewalls and network configuration schemes will pass HTTP without modification.
- Any QuickTime movie can be streamed using HTTP. QuickTime supports RTP streaming of video, audio, text, and MIDI. To stream a movie with other media types, such as sprites, you should use HTTP.

Multimedia Streaming

Early versions of QuickTime supported unicast streaming of whole movie files using HTTP.

QuickTime now supports Realtime Transport Protocol (RTP), which can be used for multicasts and for transmission of live content, as well as unicast of stored movies. QuickTime also provides the ability to stream individual tracks via RTP, allowing developers to incorporate live or remotely-stored content in a local movie.

QuickTime supports using RTSP for streaming control. QuickTime includes client software that can receive multicasts directly from routers. This software is standards-based and is interoperable with products such as VIC, VAT, or Cisco's IP/TV. QuickTime understands SDP files, which are used to "tune into" multicast streaming sessions.

QuickTime supports RTP streaming of audio, video, MIDI, text (including HREF tracks), and tweens. RTP streaming of other media types is not supported, but movies with other media types can incorporate streaming content.

QuickTime supports IETF **standard payload types** for RTP:

- Video
- H.261
- H.263+
- H.264
- DVI
- JPEG
- Audio
- Qualcomm QCELP
- GSM (receive only)
- Raw audio
- a-law

QuickTime supports **QuickTime in RTP packing** for all QuickTime encodings of

- video
- audio
- text
- MIDI
- tween

Special packing is included for optimized transmission of some QuickTime codecs, including

- Sorenson video
- Qualcomm Purevoice audio
- QDesign music

If you have your own codec, you can design special packing for it by writing a packetizer and a reassembler, as described later in this chapter.

The QuickTime File Format supports **hint tracks,** which simplify the process of packetizing QuickTime movies into RTP streams. Hint tracks allow QuickTime movies to be served from RTP servers without requiring the servers to have QuickTime software installed or to know about QuickTime media types or codecs.

QuickTime includes the ability to export QuickTime movies as hinted movies that can be streamed over RTP.

Note: QuickTime does not support RTP streaming of track references. Some QuickTime features, such as effects, chapter lists, and various applications of tweens, make use of track references. These features can be included in client movies that incorporate RTP streaming content.

Using QuickTime Streaming

This chapter describes from a programming perspective how to create, send, and receive streamed movies. Code samples for receiving and creating streaming movies are included.

Receiving Streaming Movies

An application receives streaming content by opening a movie and playing it.

Opening a Streaming Movie

In general, opening a streaming movie is like opening any QuickTime movie. You can open a streaming movie by opening

- a movie file that contains streaming tracks
- an SDP file
- a URL

You can open a movie file that contains streaming tracks, or open an SDP file, by calling `NewMovieFromFile` in the usual way.

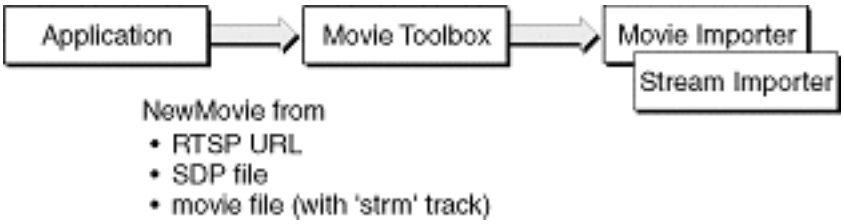
You open a movie from a URL by calling `NewMovieFromDataRef` with a URL data reference. The URL for a real-time streaming movie will use RTSP protocol. The following is a code sample for opening a movie from an RTSP URL:

```
char url[] = "rtsp://www.mycompany.com/mymovie.mov";
Handle urlDataRef;

urlDataRef = NewHandle(strlen(url) + 1);
if ( ( err = MemError()) != noErr) goto bail;

BlockMoveData(url, *urlDataRef, strlen(url) + 1);

err = NewMovieFromDataRef(&movieInfo->theMovie, newMovieActive,
    nil, urlDataRef, URLDataHandlerSubType);
DisposeHandle(urlDataRef);
```

It's also possible to open a movie file from an HTTP or FTP URL, and it's possible for that movie file to contain streaming tracks.

If you open a streaming movie from a URL or an SDP file, as shown in Figure 2-1, QuickTime will call the appropriate movie importer. If you open a movie file that contains streaming tracks, stream importers will be called.

Figure 2-1 Opening a streaming movie



Opening a streaming movie typically takes more time than opening a movie with purely local content. Each track in the movie on the server is transmitted as an RTP stream, so the client computer must establish a network connection with the server for each track, and often must establish a connection for RTSP control as well. This takes time, particularly if a dial-up connection must be established.

It also means that QuickTime can return connection status messages or networking errors in the process of opening a movie.

Playing A Streaming Movie

Applications that can already play QuickTime movies need to observe these cautions to play real-time streaming movies reliably:

- Open movies with high-level Movie Toolbox calls or a movie importer.
- Do not assume that the track structure of the movie you play reflects the track structure of the original movie; a streaming track can contain sound, video, text, MIDI, or all of these.
- Use a movie controller to play the movie, or use the `PrePrerollMovie` function to set up any streams before playing the movie.
- Display status messages returned by the movie controller.
- Be prepared to deal with network errors when your application plays a movie (see "Common Streaming Error Codes" (page 21)).
- Be prepared for movie characteristics to change dynamically; the height, width, duration, and presence of audio or video can all change as the movie plays.
- Do not assume that the movie will begin immediately.

The following sections describe some of these factors in more detail.

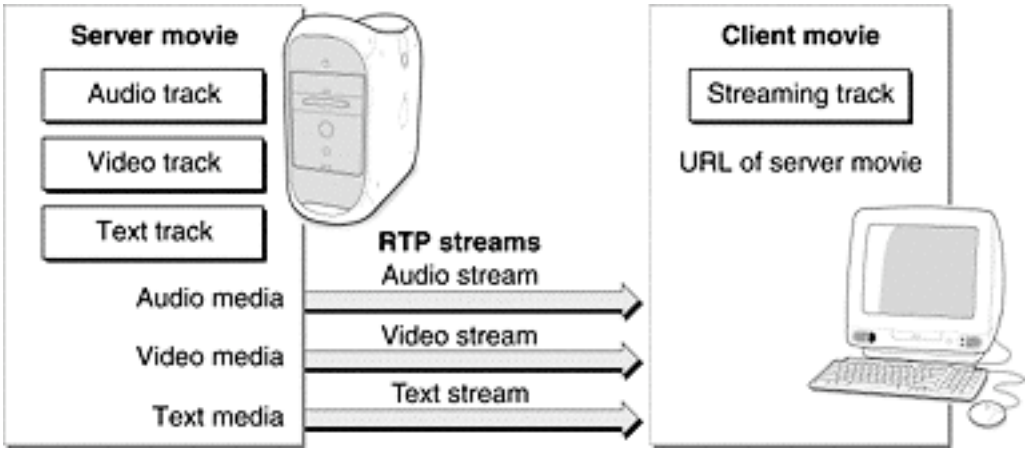
Track Structure

Unlike other QuickTime movies, streaming movies consist of two distinct movie files, one on the server and the other on the client machine, that often have different track structures. You sometimes need to distinguish between the **server movie** and the **client movie.** See Figure 2-2.

The media of each track in the server movie is transmitted as a separate RTP stream. On the client side, multiple RTP streams can be combined into a single streaming track.

If you open a movie from a URL, for example, the movie importer may create a client movie that contains only a streaming (`'strm'`) track. Unlike most other QuickTime track types, a streaming track can contain multiple media streams of different types. A streaming track in a client movie may contain the URL of a server movie with audio, video, text, and/or MIDI tracks.

Figure 2-2 Server movie and client movie



The client movie file never contains audio or video media from the server movie; they are displayed and discarded. If a client movie is saved, the saved movie contains information such as the URL of the server movie and the currently-displayed movie time. The client movie can also contain local tracks and local media, but the media samples in the server movie remain on the server except when playing.

Pre-Prerolling

Before any movie is played, QuickTime needs to allocate buffers and open appropriate media handlers. This process is called **prerolling** the movie. Before a streaming movie can be played, additional steps need to be taken, such as establishing RTP streams between the client and the server. This setup process is called **pre-prerolling.** Pre-prerolling is performed automatically when a streaming movie is played using a movie controller. If your application uses movie controllers to play movies, you do not need to take any special steps to pre-preroll a streaming movie.

If you are playing movies using lower-level commands, you will need to use the `PrePrerollMoviefunction` to set up the network connections for a streaming movie before you can preroll or play the movie. The `PrePrerollMovie` function does nothing unless a movie contains streaming content, so it's safe to call it for all movies. Here is a code sample:

```
PrePrerollMovie(myMovie, 0, GetMoviePreferredRate(myMovie),
NewMoviePrePrerollCompleteProc(MyMoviePrePrerollCompleteProc),
(void *)0L);
```

`PrePrerollMovie` operates either synchronously or asynchronously, depending on whether you specify a completion procedure when you call it.

If called asynchronously, it returns almost immediately, even if the movie contains streaming content. This allows your application to perform other tasks while awaiting the completion of the pre-prerolling. You need to call `MoviesTask` periodically to grant time for the task of pre-prerolling when using it asynchronously.

When the pre-prerolling is finished, `PrePrerollMovie` calls the completion procedure whose address you pass in the `NewMoviePrePrerollCompleteProc` parameter. In the simplest case, the completion procedure will want to preroll and start the movie.

If no completion procedure is specified, `PrePrerollMovie` returns when the pre-preroll process is complete.

Reacting to Changes in Movie Characteristics

One feature of streamed movies is that their characteristics may change dynamically during playback. For example, when you open a movie from a URL you may not know the actual height and width of the movie, its duration, how many streams it contains, whether it has a sound track, or whether it is an audio-only movie.

QuickTime will assign default values to these characteristics if they are unknown. QuickTime can notify your application when the movie characteristics become known or are changed.

In most cases, your application will want to adjust the size of the window or pane that contains the movie to reflect such changes. You make these adjustments by implementing a movie controller action filter procedure.

To do this, you first need to indicate that you want to be informed of any changes. You do this by setting a movie playback hint:

```
SetMoviePlayHints(myMovie, hintsAllowDynamicResize,
        hintsAllowDynamicResize);
```

Changes in the movie size are announced to your filter procedure by the `mcActionControllerSizeChanged` selector. Changes in other movie characteristics are announced through the `mcActionMovieEdited` selector.

Size Changes

Whenever the size of the movie changes, the associated movie controller sends the `mcActionControllerSizeChanged` action to your movie controller action filter procedure. You can intercept that action and respond to it as follows:

```
pascal Boolean MyMCActionFilterProc (MovieController theMC, short theAction,
void *theParams,
long theRefCon)
```

```
{
    Movie   myMovie = MCGetMovie(theMC);
        switch (theAction) {
            // handle window resizing
            case mcActionControllerSizeChanged:
                MyResizeWindow(myMovie);
                break;
            default:
                break;
        }
        return(false);
}
```

Duration Changes

The duration of a streaming movie or a streaming track may not be initially known. A track or movie whose duration is not known is assigned an **indefinite duration:** `x7FFFFFF`. If you do not treat this as a special case, a streaming movie will appear to your application as a very long movie indeed.

Once the pre-preroll process is complete, QuickTime should know the actual duration of the streams. If you set an action filter procedure and call `SetMoviePlayHints`, your application will be called with the `mcActionMovieEdited` selector when QuickTime determines the actual duration.

If the movie contains live content, it may not have a specific duration. In this case, the duration remains indefinite: `x7FFFFFF`. You might want to set a timeout in your code that detects the fact that QuickTime has not adjusted the duration from `x7FFFFFF` after a few seconds of movie play. QuickTimePlayer uses such a timeout and displays a “Live Transmission” message where the slider would be for a movie with a known duration.

Sound and Video Changes

Whether a streaming movie has sound (`'ears'`) or is sound-only (no video) may not be initially known or may change dynamically. If you set an action filter procedure and call `SetMoviePlayHints`, your application will be called with the `mcActionMovieEdited` selector when the sound or video characteristics change.

Other Playback Considerations

Streaming movies only play back at a rate of 1. Other playback rates, such as playing backward, are not supported.

Common Streaming Error Codes

Here is a list of common streaming error codes. Applications that play streaming movies should deal gracefully with these errors.

```
// Streaming errors
```

```
-5400    // qtsBadSelectorErr
-5401    // qtsBadStateErr
-5402    // qtsBadDataErr
-5403    // qtsUnsupportedDataTypeErr
-5404    // qtsUnsupportedRateErr
-5405    // qtsUnsupportedFeatureErr
-5406    // qtsTooMuchDataErr

// Network errors
-5420    // connection failed (couldn't connect to the server)

// Open Transport errors, Macintosh only: -3150 to -3180, -3200 to -3285,
// plus the full range of OT error codes in file OpenTransport.h

-3150    // kOTBadAddressErr; a bad address was specified
-3170    // kOTBadNameErr; couldn't do name lookup

// Open Transport errors, Windows only
10061    // couldn't connect to server

// Streaming can return all QuickTime and Macintosh Toolbox errors.
// Common error codes returned include:

// QuickTime errors
-223    // siInvalidCompression; codec not installed
// Windows-specific errors when component is loading
-2092   // componentDllEntryNotFoundErr

// RTSP errors
// Most of these error codes are the same as for HTTP.
// These errors are usually generated from the server.
3xx    // redirect
400    // bad request
401    // unauthorized
402    // payment required
403    // forbidden
404    // not found (specified movie not found on server side)
405    // method not allowed
406    // not acceptable
407    // proxy authentication required
408    // request time-out
409    // conflict
410    // gone
411    // length required
412    // precondition failed
413    // request entity too large
414    // request URI too large
415    // unsupported media type
451    // parameter not understood
452    // conference not found
453    // not enough bandwidth
454    // session not founds
455    // method not valid in this state
456    // header field not valid for resource
457    // invalid range
458    // parameter is read only
459    // aggregate operation not allowed
460    // only aggregate operation allowed
```

```
461    // unsupported transport
462    // destination unreachable
500    // internal server error
501    // not implemented
502    // bad gateway
503    // service unavailable
504    // gateway timeout
505    // version not supported
551    // option not supported
```

Serving Streaming Movies

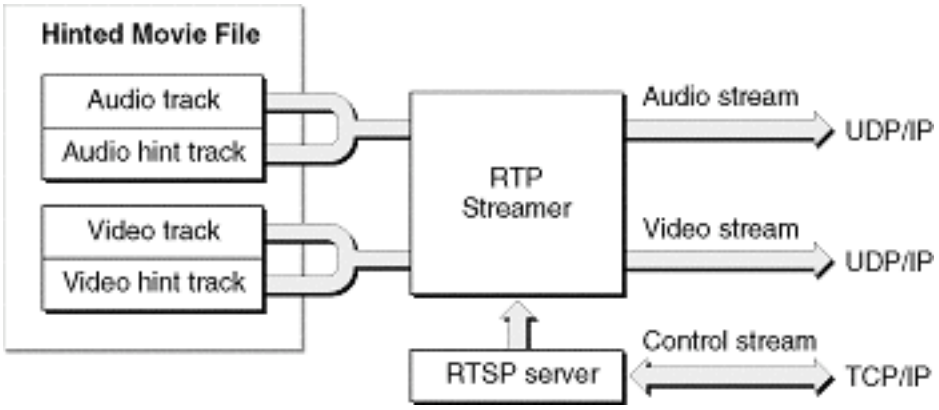
To serve QuickTime movies over RTP, your movie server must be equipped with RTP server software that understands the QuickTime file format, including the structure of **hint tracks,** which are described in the section "Hint Track Structure" (page 24). Your server also needs an RTSP controller application.

Your server does not need to have QuickTime software installed to serve streaming movies.

If your server is merely acting as a reflector for multicasts, no special software is required to reflect QuickTime movies. Forward the RTP streams on request in the usual way.

Your server uses the hint tracks in a streaming QuickTime movie to packetize the movie's media into RTP streams. Hint tracks are added to QuickTime movies to prepare them for streaming over RTP. One hint track is added to the movie for each track whose media will be streamed, as illustrated in Figure 2-3.

Figure 2-3 Streaming a hinted movie



If your server is sending a unicast of a hinted movie, the QuickTime movie controller will provide the client with the ability to pause the movie and to skip backward and forward in movie time. This will be communicated to your server over RTSP.

The RTP server does not need to know about QuickTime media types or codecs. The hint tracks within the movie file provide the information needed to turn QuickTime media into RTP packets. Each hint track contains the data needed to build packet headers for a specific track's media. The hint track also supplies a pointer to the media data that goes into each packet.

The RTP server needs to be able to parse a QuickTime movie file sufficiently to find each hint track, then to find the track and sample data that the hint track points to. The hint track contains any precalculated values that may be needed, making it easier for the server to create the RTP packets.

Hint tracks offload a great deal of computation from the RTP server. Consequently, you may find that an RTP server is able to send data more efficiently if it is contained in a QuickTime movie, even if the RTP server already understands the media type, codec, and RTP packing being used.

For example, the H.263+ video codec is an IETF-standard RTP format which your server may already understand, but creating an H.263+ stream from video data requires complex calculations to properly determine packet boundaries. This information is precalculated and stored in the hint track when H.263+ video is contained in a QuickTime movie, allowing your server to packetize the data quickly and efficiently.

If you are writing QuickTime extensions to an RTP server application, you will need to read the *QuickTime File Format* documentation, as well as "Hint Tracks" (page 24) in this document.

Hint Tracks

You prepare a QuickTime movie for RTP streaming by adding hint tracks. Hint tracks allow an RTP server to stream QuickTime movies without requiring the RTP server to understand QuickTime media types, codecs, or packing. The RTP server needs to understand the QuickTime file format sufficiently to find tracks and media samples in a QuickTime movie, however.

Each track in a QuickTime movie is sent as a separate RTP stream, and the recipe for packetizing each stream is contained in a corresponding hint track. Each sample in a hint track tells the RTP server how to packetize a specific amount of media data. The hint track sample contains any data needed to build a packet header of the correct type, and also contains a pointer to the block of media data that belongs in the packet.

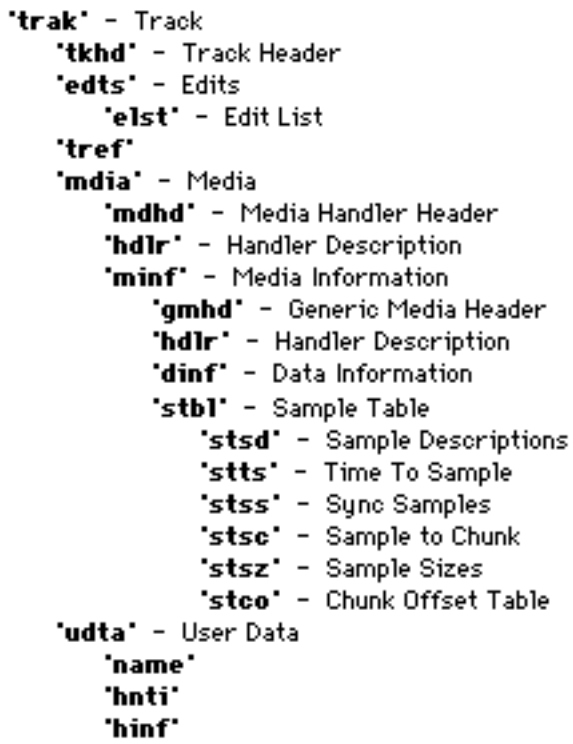
There is at least one hint track for each media track to be streamed. It is possible to create multiple hint tracks for a given track's media, optimized for streaming the same media over networks with different packet sizes, for example. The hinter included in QuickTime creates one hint track for each track to be streamed.

Hint tracks are structured in the same way as other QuickTime movie tracks. Standard QuickTime data structures, such as track references, are used to point to data. In order to serve a QuickTime movie, your RTP server must locate the hint tracks contained in the movie and parse them to find the packet data they point to. You should review the *QuickTime File Format* documentation before you read the remainder of this hint track information. You will also find it helpful to refer to the *QuickTime File Format* periodically in the course of reading this document.

Hint Track Structure

Hint tracks are atoms of type `'trak'`. A hint track atom contains a track header atom (`'tkhd'`), an edits atom (`'edts'`), a track reference atom (`'tref'`), a media atom (`'mdia'`), and usually a track user data atom (`'udta'`). The atoms are normally present in the order described, but this is not a requirement. Most of these atoms have child atoms. An expanded diagram of a typical hint track atom is shown in Figure 2-4.

Figure 2-4 A typical hint track atom



A detailed description of each atom's contents follows. This document focuses on atom contents that are specific to hint tracks. Refer to the *QuickTime File Format* documentation for generic information about the structure and contents of the atom types found in track atoms.

Note: The actual hint samples are not part of the hint track atom structure. The hint track atom structure contains the information you need to locate and interpret the samples. A description of hint samples follows the description of the hint track's atoms.

Track Header Atom

The `flags` field of the track header atom must be 0x000000, indicating the track is inactive, and not part of the movie, preview, or poster. Hint tracks must be marked as inactive for the movie to play locally.

The creation time, modification time, and track ID fields are set as they would be for any track.

The `duration` field is also the duration of the media track being hinted, expressed in movie time scale.

The `layer` field is not used.

The alternate group field can be used to distinguish alternate language or alternate data rate tracks in streaming movies. It is not required that all servers support this feature. It is not recommended that applications attempt to make use of this feature at this time. This field is normally set to 1.

The remaining track header fields are not used for hint tracks.

Unused fields should be set to 0 when creating hint tracks and ignored when using them.

Edits Atom

The edits atom contains an edits list atom (`'elst'`). Edit lists can be used for hint tracks as they can for any other track type.

The fields of the edits list atom (`'elst'`) are used as follows:

The `flags` field is not used, and should be set to 0.

The `numEntries` field is the number of list entries. Unless the hint track has been edited, it will have either 1 or 2 entries.

Each list entry consists of a duration, a media time, and a rate.

If the track should begin playing after the movie begins, there is an "empty edit" whose duration is the amount of time that passes until the track begins playing, whose media time is -1, and whose media rate is 1. No data should be sent for this track until the number of movie time units specified by the duration field has passed.

Whether or not there is an empty edit, there will typically be one entry whose duration is the media duration in movie timescale units, whose media time is 0, and whose rate is 1.

Track Reference Atom

Each hint track refers to a media track. The hint track contains a track reference atom (`'tref'`) that contains exactly one child atom of type `'hint'`. This child atom holds the track ID of the media track. The whole track reference atom has this structure:

Atom Type	Field	Bytes	Description
Track Reference			Contains a `'hint'` atom
	`Size`	4	Size of track reference atom (including `'hint'` atom); 32-bit integer.
	`Type`	4	`'tref'`
Track Hint			Contains the track ID of the media track being hinted
	`Size`	4	Size of this child atom; 32-bit integer.
	`Type`	4	`'hint'`
	`TrackID`	4	Track ID of the media track being hinted; 32-bit integer.

The Track ID field of the 'hint' atom contains the track ID of the media track being hinted. The target track ID can be found in the media track's track header atom ('tkhd'). All media sample data should be taken from the specified media track.

There could theoretically be a list of track IDs for a hint track that hinted multiple media tracks, but the current hinter only references one media track per hint track.

Creating Streaming Movies

Streaming movies come in two forms: server movies and client movies.

You create a server movie that can be streamed over RTP by adding hint tracks. Hint tracks tell the server how to packetize the movie. You add hint tracks by exporting a movie as a hinted movie using QuickTime's standard movie export mechanism.

You create a client movie that includes streaming content by adding one or more streaming tracks. A streaming track tells the client where to get the streaming media. In its simplest form, a client movie consists of just a streaming track containing the URL of a movie on a server. A client movie can contain multiple streaming tracks.

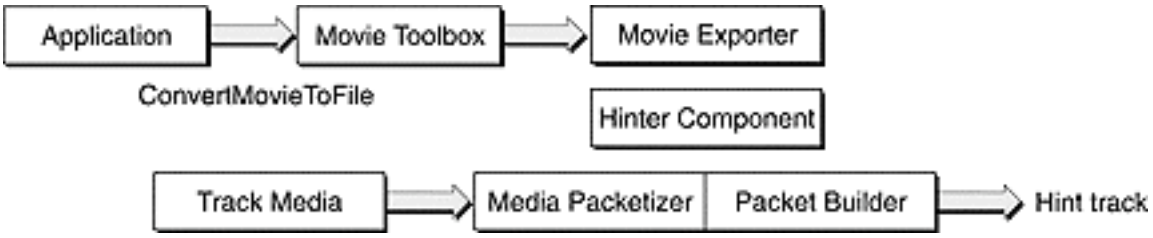
A client movie can contain non-streaming tracks with local media content as well as streaming tracks. Streaming tracks can be composited with local tracks. For example, a streaming track could be used as the source for an effect track that is local to the client movie.

Server Movies

You create a streaming movie for an RTP server by adding hint tracks to an existing movie. You do this by calling ConvertMovieToFile, which invokes a movie exporter component. This displays a standard dialog box that lets the user specify "Export Movie to Hinted Movie," set the parameters for hinting the movie, select compressors, and specify a file name and directory for the hinted movie (see Figure 2-5).

The hinting is performed by media packetizer components. QuickTime selects an appropriate media packetizer for each track and routes each packetizer's output through an Apple-provided packet builder to create a hint track. One hint track is created for each streamable track in the movie.

Figure 2-5 Exporting a hinted movie



Hint tracks are quite small compared with audio or video tracks. A movie that contains hint tracks can be played from a local disk or streamed over HTTP, like any other QuickTime movie. The hint tracks are only used when streaming the movie over RTP.

As long as your application supports movie exporter components, it should be able to create hinted movies. If you want to bypass the standard dialog for user input, selecting "Export Movie to Hinted Movie" programmatically, you will need to modify your code by adding the appropriate selectors.

A hinted movie does not need to be self-contained (flattened). It can reference sample media contained in other files. But observe these cautions:

- Use file names that are transportable between the system that the movies are created on and the RTP server.
- The relative path from the movie file to the data files must not change after the movie is saved.
- The simplest way to ensure both of these is to use only lowercase letters in file names, without spaces, and to keep the media data files and the movie file in the same folder or directory.
- Hint tracks should be created as the last step in making a streaming movie. Any editing of the movie that adds or deletes sample data, including the flattening of a movie with edit lists, invalidates the hint track. If your application edits a hinted movie in a way that invalidates the hint tracks, delete the hint tracks and re-export the movie.

Hint tracks are marked as inactive so they do not interfere with local playback. If you call `FlattenMovie` with the `flattenActiveTracksOnly` flag, the hint tracks are deleted from the flattened movie.

This release of QuickTime supports RTP streaming of video, audio, text (including HREF Tracks), and MIDI. If your movie contains other media types, or features that rely on track references, you cannot currently export the entire movie to a hinted movie. You can either stream such movies over HTTP, or you can put some of the tracks into a client movie and stream the rest, as described in the section "Compositing Streaming and Non-Streaming Tracks" (page 29).

Client Movies

You incorporate streaming content in a client movie by adding at least one streaming track. The simplest form of client movie has only one track: a streaming track with the URL of a movie on a server.

A streaming track has a media type of `kQTSStreamMediaType` (`'strm'`) and contains a single media sample: typically either an RTSP URL of a streaming movie or the SDP text describing a multicast.

The code example in Listing 2-1 shows how to create a streaming track, insert a sample description, and add a URL media sample.

Listing 2-1 Creating a streaming track with an RTSP URL

```
Handle dataRef;
long dataLength;
char url[] = "rtsp://myserver.bigcompany.com/mystreaming.mov";
dataRef = NewHandle(strlen(url) + 1);
BlockMoveData(url, *urlDataRef, strlen(url) + 1);
dataLength = GetHandleSize(dataRef);
newMedia = NewTrackMedia(newTrack, kQTSStreamMediaType,
        kQTSMediaTimeScale, handleDataRef, HandleDataHandlerSubType);
err = BeginMediaEdits(newMedia);
qtsDesc =
(QTSSampleDescriptionHandle)NewHandleClear(sizeof(QTSSampleDescription));
(**qtsDesc).descSize = sizeof(QTSSampleDescription);
```

```
(**qtsDesc).dataFormat = 'rtsp';
(**qtsDesc).dataRefIndex = 1;
(**qtsDesc).version = kQTSSampleDescriptionVersion;
duration = kQTSInfiniteDuration;
err = AddMediaSample(newMedia, dataRef, 0, dataLength, duration,
        (SampleDescriptionHandle)qtsDesc, 1, 0, nil);
err = EndMediaEdits(newMedia);
err = InsertMediaIntoTrack(newTrack, 0, 0, GetMediaDuration(newMedia),
        kQTSNormalForwardRate);
```

A streaming track in a client movie can point to a server movie containing audio, video, text, and MIDI tracks. Any or all of the tracks in the server movie can appear as a single streaming track in the client movie.

To sum up:

- A client-side streaming movie contains at least one streaming `'strm'` track.
- A streaming track contains a single media sample, typically an RTSP URL that points to streaming content. It can also contain SDP information for a multicast.
- The streaming content can be a live stream or a stored movie on a streaming server.
- The movie on the server can contain any number of tracks; multiple tracks in the server movie may be represented in the client movie as a single streaming track.

Compositing Streaming and Non-Streaming Tracks

A client movie can contain non-streaming tracks with locally-stored media, in addition to one or more streaming tracks. Use this technique to add a live stream to a locally-stored movie, or to distribute a movie on CD-ROM that references the latest version of some media stored on a server.

You can composite a streaming track with local tracks. A streaming track can be positioned in time and space like any other track. Bear in mind that the streaming track may contain more than one video, audio, text, or other streams, however.

If your application creates movies, you will want to provide options for creating a server movie and a client movie from the same data, as well as providing options for putting some tracks in the server movie and some in the client movie.

Ordinarily, you create a streaming movie by exporting a movie as a hinted movie, putting that hinted movie on an RTP server, and creating a client movie that contains the URL of the movie on the server.

Sometimes, however, you will want to export part of an existing movie to a hinted server movie while incorporating other parts in the client movie. You might do this to stream the audio and video parts of a movie over RTP, while streaming wired sprites or chapter lists over HTTP.

For example, to create a streaming movie with a chapter list, extract the text track containing the chapter list from the movie and export the remainder as a hinted movie. Create a client movie with just a streaming track containing the URL of the hinted movie. Add the extracted text track to the client movie and make it a chapter list by adding a track reference to the streaming track. The client movie now contains:

- a local text track containing the chapter list

- a streaming track that points to the rest of the movie
- a track reference of type `'chap'` in the streaming track that references the chapter list track

The server movie contains the original movie, minus the chapter list, plus hint tracks.

You would use a similar technique to stream a movie with QuickTime video effects, storing the effects tracks in the client movie and applying the track references to the streaming tracks that act as sources to the effects.

Media Packetizers

Media packetizers are components that understand how to break QuickTime media into packets for RTP transmission. Packet reassemblers are components that understand how to put those packets back together to reconstitute the media. Packetizers and reassemblers are written in pairs; one to packetize for transmission, the other to reassemble during reception. Media packetizers are used during live transmissions and during the creation of hint tracks. Packet reassemblers are used when receiving streaming content.

RTP transmission is a best-effort delivery system. Over the Internet, it is inherently lossy. Packets may be dropped and not retransmitted; they may arrive out of order or after substantial delay. For streaming to work under these conditions, the sender and receiver must deal intelligently with unpredictable loss. In QuickTime, this intelligence resides primarily in the packetizers and reassemblers.

For example, a packetizer may break a video frame into a grid of independently coherant rectangles. If a packet is lost, only one rectangle needs to be discarded; the rest of the video frame can be reconstructed. The reassembler in this case might substitute the old rectangle for that part of the grid when a packet is lost, or slowly fade that rectangle to black, or use some other recovery scheme.

QuickTime includes a generic packetizer/reassembler pair that works with most QuickTime content. It also includes specialized packetizer/reassembler pairs optimized for specific media types and compression formats, such as Sorenson video, Qualcomm Purevoice audio, and QDesign music.

Writing Media Packetizers

You can write your own packetizer/reassembler pairs and add them to QuickTime. You might want to do this if you have written your own codec, for example, or if you have a particularly clever scheme for packing media or recovering from packet loss.

Passing Non-Media Data

Media packetizers and packet reassemblers are generally concerned with media sample data, such as video frames or audio samples. QuickTime movies commonly contain track-level information, such as a transformation matrix, or audio volume, that is applied to the sample data but is not embedded in it.

Packetizers and reassemblers can optionally support the passing of this type of data. If your packetizer supports passing of a particular type of information, such as audio volume or transformation matrix, it should indicate this in its public resource (as described in "The 'pcki' Public Resource" (page 32)).

In the course of normal operations, a media packetizer may receive a `SetInfo` command that specifies this type of information. A packetizer that can send a transformation matrix, for example, could receive such a call whenever the matrix changes. It is up to the packetizer to package this information in a way that will be recognized by the reassembler and to send it along.

When a reassembler receives this type of information, it signals the fact by calling `RTPRssmSendStreamHandlerChanged`. It may then receive a series of `GetInfo` calls to determine what information it has. It passes the information in response to these calls, and QuickTime does the rest.

The 'pcki' Public Resource

A packetizer must provide a public resource of type `'pcki'`. The public resource contains information about the capabilities of a given packetizer. This information lists the media types and compression formats the packetizer can work with. It also lists the track characteristics the packetizer can work with, such as layers or transformation matrices. In addition, it provides information about the packetizer's performance characteristics, such as its speed or ability to recover from packet loss.

QuickTime selects the packetizer best suited to the media, compression format, and track characteristics. If there are multiple packetizers that can work with a given track, performance is considered.

For example, one packetizer might be able to pack H.261-compressed video, but be unable to handle any transformation matrix other than identity. Another packetizer might also do H.261 and be able to handle any kind of transformation matrix. If a matrix was being used to scale and translate the media, QuickTime would select the more versatile packetizer. If there were no matrix applied to the track, QuickTime might select the faster packetizer for a live transmission, and the packetizer that dealt best with loss when creating a hint track.

A movie that contains streaming tracks can be played from a local disk or served via HTTP. You can create a client movie with a streaming track that contains the URL of a server movie, then embed the client movie in a website or distribute it on a CD-ROM. When the movie is played, the user's computer will establish a network connection to the specified server for each streaming track.

The format of the public resource is defined in `QTStreamingComponents.r`. It is shown in Listing 3-1.

Listing 3-1 Format of the 'pcki' resource

```
type 'pcki' {
    hex longintmediaType;
    hex longintdataFormat;
    hex longintvendor;
    hex longintcapabilityFlags;
    byte    canPackMatrixType;
    byte = 0;
    byte = 0;
    byte = 0;
    longint = $$CountOf(characteristicArray); /* Array size*/
    array characteristicArray {
        hex longinttag;
        hex longint value;
    };
```

```
    hex longintpayloadFlags;
    byte    payloadID;      /* if static payload */
    byte = 0;
    byte = 0;
    byte = 0;
    cstring;
};
```

The first three fields describe the media type, data format (compression format), and the manufacturer. The media type and data format are matched against the given track and media information when deciding whether to use a given packetizer. If your packetizer supports multiple compression formats, set `dataFormat` to 0. If it supports multiple media types, set `mediaType` to 0. The vendor field is for informational purposes only.

The `capabilityFlags` field can contain a combination of these flag constants:

```
enum {
    kMediaPacketizerCanPackEditRate = 1 << 0,
    kMediaPacketizerCanPackLayer = 1 << 1,
    kMediaPacketizerCanPackVolume = 1 << 2,
    kMediaPacketizerCanPackBalance = 1 << 3,
    kMediaPacketizerCanPackGraphicsMode = 1 << 4,
    kMediaPacketizerCanPackEmptyEdit = 1 << 5
};
```

These flags describe a packetizer's ability to deal with track characteristics. By indicating a capability, a packetizer says that it can transmit track information (such as a matrix) to the reassembler on the client side, so that it can be used in the client movie. The specific flags have these meanings:

Flag	Description
`kMediaPacketizer-CanPackEditRate`	Changing edit rates. This is usually set for video, since each video frame typically has its own timestamp. Because of this, there isn't really a constant play rate. Sound, on the other hand, does have a play rate. If your packet format doesn't allow specification of non-standard rates, do not set this flag.
`kMediaPacketizer-CanPackLayer`	Only for visual formats. Indicates that you can communicate the layering information for this stream. If your packetizer supports this feature, the layering information from the QuickTime track structure may be passed to you in an `RTPMPSetInfo` call.
`kMediaPacketizer-CanPackVolume`	Sound only. Indicates you can communicate volume information from the QuickTime track structure.
`kMediaPacketizer-CanPackBalance`	Sound only. Indicates you can communicate sound balance information from the QuickTime track structure.
`kMediaPacketizerCan-PackGraphicsMode`	Video only. Indicates you can transmit the graphics mode and `opColor` for graphics modes other than `ditherCopy` and `srcCopy`.

Flag	Description
kMediaPacketizerCan-PackEmptyEdit	Empty edits. For sound formats, this is generally set, since the Sound Stream Handler will synthesize silence for any missing packets, which is exactly the intended effect for an empty edit in a sound track. For visual tracks this requires either sending down a duration, so that the Video Stream Handler knows when the frame is supposed to end, or some other method of describing empty edits in the packet format.
canPackMatrixType	Applies only to visual tracks.

The canPackMatrixType flag is one of:

```
canPackIdentityMatrixType 0x00
canPackTranslateMatrixType 0x01
canPackScaleMatrixType 0x02
canPackScaleTranslateMatrixType 0x03
canPackLinearMatrixType 0x04
canPackLinearTranslateMatrixType 0x05
canPackPerspectiveMatrixType 0x06
```

Note that these are the same values as those defined in ImageCompression.h as return values for GetMatrixType. The value indicates you can communicate the specified matrix type.

There are two performance characteristics currently defined:

```
#define kMediaPacketizerSpeedTag          'sped'    /* 0-255, 255 is fastest */

#define kMediaPacketizerLossRecoveryTag   'loss'    /* 0-255, 0 can't handle
                                    any loss, 128 can handle 50% packet loss */
```

The speed tag is relative to other packetizers for the same media type and compression format. A value of 128 is a reasonable default.

The payloadFlags field is set to either kRTPMPPayloadTypeDynamicFlag for a dynamic payload type or kRTPMPPayloadTypeStaticFlag for a static payload type.

The payload ID field of the 'pcki' resource is set to the IETF-defined RTP payload value if a static payload type is used.

The C string contains the RTP payload type text if a dynamic payload type is used.

A code example of a resource for a packetizer that supports the 'OVAL' video compression format is this:

```
resource 'thnr' (128) {
    {
        'pcki', 1, 0,
        'pcki', 128, cmpResourceNoFlags,
    };
};
resource 'pcki' (128) {
    'vide',         // media type
    'OVAL',         // data format type
    'ABCD',         // manufacturer type
```

```
    kMediaPacketizerCanPackEditRate,
    canPackIdentityMatrixType,
    {
        kMediaPacketizerSpeedTag, 128,
        kMediaPacketizerLossRecoveryTag, 50
    },
    kRTPMPPayloadTypeDynamicFlag,
    0,
    "OVAL-49545"
};
```

This packetizer is of medium speed, can handle some loss, can pack any arbitrary play rate, but can't pack any non-identity matrix. Its RTP payload type is a dynamic identifier, identified by the string `OVAL-49545`.

Media Packetizer Functions

QuickTime Streaming calls a media packetizer during the course of hinting a movie or during live transmission. The packetizer is passed sample data, which it breaks into packets and passes to a packet builder. The packet builder may then transmit the packets over RTP or use them to create a hint track. The media data is presented to the packetizer in the same format for a live transmission or for hinting, and the packetizer produces the same output in both cases.

The packetizer component must implement several functions, and must also provide a public component resource that describes the type of media, compression, and track characteristics that the packetizer supports. This resource also provides information on the packetizer's relative speed and the format's ability to handle loss.

The following functions can be implemented by media packetizer components. Most of these functions must be implemented in your component, but some are optional.

- `RTPMPInitialize`
- `RTPMPPreflightMedia`
- `RTPMPIdle`
- `RTPMPSetSampleData`
- `RTPMPReset`
- `RTPMPSetInfo`
- `RTPMPGetInfo`
- `RTPMPSetTimeScale`
- `RTPMPGetTimeScale`
- `RTPMPSetTimeBase`
- `RTPMPGetTimeBase`
- `RTPMPHasCharacteristic`
- `RTPMPSetPacketBuilder`
- `RTPMPGetPacketBuilder`
- `RTPMPSetMediaType`

- RTPMPGetMediaType
- RTPMPSetMaxPacketSize
- RTPMPGetMaxPacketSize
- RTPMPSetMaxPacketDuration
- RTPMPGetMaxPacketDuration
- RTPMPDoUserDialog
- RTPMPSetSettingsFromAtomContainerAtAtom
- RTPMPGetSettingsIntoAtomContainerAtAtom
- RTPMPGetSettingsAsText

Sequence of Events

When QuickTime is asked to provide a packetizer, it selects the packetizer based on the media type, data format, and other characteristics, such as whether a matrix transformation is in use. It selects the packetizer best able to handle the media and track characteristics by examining the packetizer's public resource.

Once a packetizer has been selected, it is opened. It is then asked to preflight the media, to verify that it can actually packetize the desired media data. If the packetizer indicates that it can handle the media, it is initialized. The packetizer then receives a series of setup calls required to prepare it for operation. These calls deliver information such as the media time scale and the packet builder to use for output.

Once setup is complete, the packetizer receives a series of calls with sample data. If the packetizer can process the data immediately, it does so; otherwise it returns a flag that indicates it is still processing the data. In the latter case, the packetizer's `Idle` function is called perodically until the packetizer has completed its processing. When it is ready, the packetizer creates packets by making calls to a packet builder component.

The packetizer is then called again with more sample data. This continues until all the media data has been packetized.

At any time in this process, the packetizer can be asked to return information, or to flush its input buffer, or to reset itself and prepare for a new sequence.

Packetizer Component Type and Subtype

Packetizers have a component type of `kRTPMediaPacketizerType ('rtpm')`. The subtype can be any four-character combination. Note, however, that all-lowercase types are reserved by Apple.

Media Preflight

Once QuickTime has selected and opened a packetizer component, it will call the packetizer's `RTPMPPreflightMedia` function to verify that the packetizer can handle the specific media and sample description. A packetizer can reject media as a result of this call, even if the media fits the profile described in the packetizer's public resource.

For example, the `'pcki'` resource for the Qualcomm PureVoice audio packetizer indicates support for audio media and PureVoice compression, but the packetizer only supports 8 kHz sample rates over RTP. The `RTPMPPreflightMedia` call to this packetizer returns `noErr` for 8 kHz audio, and an error for any other sample rate.

The media preflight function is defined as follows:

```
pascal ComponentResult RTPMPPreflightMedia(RTPMediaPacketizer rtpm,
                OSType inMediaType,
                SampleDescriptionHandle inSampleDescription);
```

This call is made to make sure your packetizer can handle a given media type and sample description. Return `noErr` if you can support it. Return `qtsUnsupportedFeatureErr` if you cannot.

Initialization

If your packetizer returns `noErr` during the media preflight, it will be initialized before it is asked to handle any data. The initialization function is defined as follows:

```
pascal ComponentResult RTPMPInitialize(RTPMediaPacketizer rtpm,
                SInt32 inFlags);
```

The value of `inFlags` can be 0 or `kRTPMPRealtimeModeFlag = 0x00000001`.

The `kRTPMPRealtimeModeFlag` flag indicates that your packetizer is being used for live transmission, rather than for hinting. You might use this information if your packetizer has a tradeoff between speed and fidelity.

Setup and Information Functions

Once your packetizer is initialized, it will receive several calls to set the information it needs prior to packetizing data, such as the media timescale. It may also be called with requests to return the settings it has been given. The main setup functions are these:

- `RTPMPSetTimeScale`
- `RTPMPGetTimeScale`
- `RTPMPSetPacketBuilder`
- `RTPMPGetPacketBuilder`
- `RTPMPSetMediaType`
- `RTPMPGetMediaType`
- `RTPMPSetMaxPacketSize`

- RTPMPGetMaxPacketSize
- RTPMPSetMaxPacketDuration
- RTPMPGetMaxPacketDuration

The Set functions listed above are used to set the time scale, packet builder (which receives your output), media type, maximum packet size, and maximum packet duration. These functions will be called before you are asked to begin packetizing data, and will not be called after you have begun packetizing data.

The Get functions listed above can be called at any time. When your packetizer is called with one of these Get functions, return the data that was passed to you in the corresponding Set function.

- RTPMPSetTimeBase
- RTPMPGetTimeBase

The time base functions may be called for live transmission. The Set function sets the QuickTime time base that is in use. Your packetizer can query this time base to find out what time it is in the live stream. Your packetizer should not rely on receiving this call.

RTPMPHasCharacteristic may be called to determine whether your media packetizer has a particular characteristic, such as whether it supports a user settings dialog. Return qtsBadSelectorErr if your packetizer does not have the given characteristic.

```
ComponentResult RTPMPHasCharacteristic (
                    RTPMediaPacketizer rtpm,
                    OSType inSelector,
                    Boolean *outHasIt);
```

Parameter	Definition
rtpm	The component instance of your media packetizer
inSelector	A selector for the characteristic
outHasIt	Return a boolean which is true if your media packetizer has this characteristic, false otherwise.

The inSelector parameter may have these values:

Value	Definition
kRTPMPNoSampleDataRequired-Characteristic	If set, the media packetizer does not require the actual sample data to perform packetization. The caller can pass in nil data pointers instead of pointers to the actual media data (see RTPMPSetSampleData).
kRTPMPHasUserSettingsDialog-Characteristic	If set, the media packetizer supports the calls RTPMPDoUserDialog, RTPMPGetSettingsInto-AtomContainerAtAtom, and RTPMPSetSettingsFrom-AtomContainerAtAtom.

Value	Definition
kRTPMPPrefers-ReliableTransport-Characteristic	If set, the packetizer would prefer its data to be sent reliably (such as through text or music tracks).
kRTPMPRequiresOut-OfBandDimensions-Characteristic	If set, the original visual dimensions of the media data cannot be determined simply by looking at the media packets, and must be transmitted via some other method.

If your packetizer supports a user dialog or additional settings, you may also receive these dialog or settings calls as part of the set up process:

```
RTPMPDoUserDialog
RTPMPSetSettingsFromAtomContainerAtAtom
RTPMPGetSettingsIntoAtomContainerAtAtom
RTPMPGetSettingsAsText
```

- The `RTPMPDoUserDialog` function will invoke your packetizer's modal dialog to obtain user settings.
- The `RTPMPGetSettingsIntoAtomContainerAtAtom` function expects you to return those settings into an atom container at the specified offset.
- The `RTPMPSetSettingsFromAtomContainerAtAtom` function is used to set the user settings programmatically, bypassing the user dialog.
- The `RTPMPGetSettingsAsText` function expects you to return your user settings as text.

In addition to the set up calls listed above, you may receive `RTPMPSetInfo` function calls. These are used to send a variety of information to your packetizer, as indicated by the selector. If you don't support a given selector, return `qtsBadSelectorErr`.

The `RTPMPSetInfo` selectors can be:

```
kQTSSourceTrackIDInfo ('otid'),         /* UInt32* */
kQTSSourceLayerInfo ('olyr'),           /* UInt16* */
kQTSSourceLanguageInfo ('olng'),        /* UInt16* */
kQTSSourceTrackFlagsInfo ('otfl'),      /* SInt32* */
kQTSSourceDimensionsInfo ('odim'),      /* QTSDimensionParams* */
kQTSSourceVolumesInfo ('ovol'),         /* QTSVolumesParams* */
kQTSSourceMatrixInfo ('omat'),          /* MatrixRecord* */
kQTSSourceClipRectInfo ('oclp'),        /* Rect* */
kQTSSourceGraphicsModeInfo ('ogrm'),    /* QTSGraphicsModeParams* */
kQTSSourceScaleInfo ('oscl'),           /* Point* */
kQTSSourceBoundingRectInfo ('orct'),    /* Rect* */
kQTSSourceUserDataInfo ('oudt'),        /* UserData */
kQTSSourceInputMapInfo ('oimp'),        /* QTAtomContainer */
```

Your packetizer can be called with these selectors even if it indicated that it couldn't handle the given characteristic in its `'pcki'` resource. In that case, return `qtsBadSelectorErr`.

Your packetizer may receive the related `RTPMPGetInfo` function at any time:

```
pascal ComponentResult RTPMPGetInfo(RTPMediaPacketizer rtpm,
               OSType inSelector, void *ioParams);
```

Your packetizer is expected to return the requested information in `ioParams`. The type of `ioParams` is dependent on `inSelector`. If you don't support the selector, return `qtsBadSelectorErr`. The selectors can be any of the selectors for `RTPMPSetInfo` or any of the additional selectors listed below:

```
/* info selectors -- get only */
kRTPMPPayloadTypeInfo ('rtpp'),                 /* RTPMPPayloadTypeParams* */
kRTPMPRTPTimeScaleInfo ('rtpt'),                /* TimeScale* */
kRTPMPRequiredSampleDescriptionInfo  ('sdsc'), /* SampleDescriptionHandle* */
kRTPMPMinPayloadSize ('mins'),                  /* UInt32*, doesn't include
                                                   rtp header; default 0 */
kRTPMPMinPacketDuration ('mind'),               /* UInt3* in milliseconds;
                                                   default is no min */
kRTPMPSuggestedRepeatPktCountInfo  ('srpc'),    /* UInt32* */
kRTPMPSuggestedRepeatPktSpacingInfo ('srps'),   /* UInt32* in milliseconds */
kRTPMPMaxPartialSampleSizeInfo  ('mpss'),       /* UInt32* in bytes */
kRTPMPPreferredBufferDelayInfo  ('prbd')        /* UInt32* in milliseconds */
```

The `kRTPMPPayloadTypeInfo` selector requires you to fill out an `RTPMPPayloadTypeParams` structure:

```
struct RTPMPPayloadTypeParams {
    UInt32  flags;
    UInt32  payloadNumber;
    short   nameLength;
/* in: size of payloadName buffer (counting null terminator);
/* this will be reset to needed length and paramErr returned if too small */
    char *  payloadName;    /* caller must provide buffer */
};
typedef struct RTPMPPayloadTypeParams RTPMPPayloadTypeParams;

/* flags for RTPMPPayloadTypeParams */
enum {
    kRTPMPPayloadTypeStaticFlag= 0x00000001,
    kRTPMPPayloadTypeDynamicFlag = 0x00000002
};
```

If you have a dynamic RTP payload type, you need to copy the payload type string to the buffer pointed to by `payloadName` (a null-terminated string). When `RTPMPGetInfo` is called, `RTPMPPayloadTypeParams` will be the available size of the input buffer (specified by `payloadName`). If this size is too small for the payload identifier, set `nameLength` to the size of the buffer you need (including the null terminator) and return `paramErr`. This will cause QuickTime to reallocate the buffer and call your packetizer again.

Constant	Definition
`kRTPMPRTPTimeScaleInfo`	Return the RTP timescale used by this packetizer if the time scale must be a specific value; otherwise, return an error for this selector.
`kRTPMPRequiredSample-DescriptionInfo`	Return a handle to a sample description specifying that only data with the given sample description is supported; if no such sample description exists, return an error for this selector.
`kRTPMPMinPayloadSize`	Return the minimum payload size, in bytes, of packets allowed by this packetizer (`UInt32`).
`kRTPMPMinPacketDuration`	Return the minimum packet duration allowed by this packetizer (`UInt32`, in milliseconds).

Constant	Definition
kRTPMPSuggested-RepeatPktCountInfo	Return the suggested number of repeat packets to send for this media (UInt32). This will typically be 0 for audio or video, nonzero for text or MIDI.
kRTPMPSuggested-RepeatedPktSpacingInfo	Return the suggested temporal distance between repeat packets (UInt32, in milliseconds).
kRTPMPMaxPartial-SampleSizeInfo	Return the size of the largest partial sample your packetizer can accept (UInt32, in bytes). If your packetizer returns true for HasCharacteristic (kRTPMPPartialSamplesRequired-Characteristic), this call will be made to ask what the largest size (in bytes) is that the packetizer can handle receiving in a SetSampleData call. Very few packetizers will need to set this; it is intended only for media formats where a single media sample can be huge. MPEG is an example; one media sample covers the entire span of the MPEG movie.
kRTPMPPreferred-BufferDelayInfo	Return the preferred buffer delay for your packetizer (UInt32, in milliseconds). Most packetizers will not need to set this. This would only be set if the packet duration was sufficiently long that you needed a longer buffer delay to work.

Sample Processing Functions

Once setup is complete, QuickTime will begin calling your packetizer's RTPMPSetSampleData function:

```
pascal ComponentResult RTPMPSetSampleData(RTPMediaPacketizer rtpm,
                const RTPMPSampleDataParams *inSampleData, SInt32 *outFlags);
```

This is where most of a typical packetizer's work is done. Your packetizer is presented with a block of media sample data, you break it into packets according to your own algorithm, and you pass the results to the selected packet builder.

- If your packetizer works synchronously, you packetize the data and return 0 in outFlags. If you need more sample data to complete a packet, return kRTPMPWantsMoreDataFlag.
- If your packetizer works asynchronously, you set outFlags to kRTPMPStillProcessingData and your packetizer continues its work in the RTPMPIdle function, which will be called periodically. RTPMPIdle should also set outFlags to kRTPMPStillProcessingData if it still has work to do. It sets outFlags to 0 if all the sample data has been processed.
- If your packetizer works synchronously, RTPMPIdle always sets outFlags to 0.

Your media packetizer must call the release procedure when done with the media data.

Do the processing work in the RTPMPSetSampleData function if it does not take up too much CPU time; otherwise, do it in your idle function:

```
struct RTPMPSampleDataParams {
    UInt32                  version;
    UInt32                  timeStamp;
```

```
    UInt32                  duration;           /* 0 if not specified*/
    UInt32                  playOffset;         /* 0 if not specified*/
    Fixed                   playRate;
    SInt32                  flags;
    UInt32                  sampleDescSeed;
    Handle                  sampleDescription;
    RTPMPSampleRef          sampleRef;
    UInt32                  dataLength;
    const UInt8 *           data;
    RTPMPDataReleaseUPP     releaseProc;
    void *                  refCon;
};
typedef struct RTPMPSampleDataParamsRTPMPSampleDataParams;

/* flags for RTPMPSampleDataParams*/
enum {
    kRTPMPSyncSampleFlag= 0x00000001
};
```

Field	Definition
`version`	Version of the data structure. Currently always 0.
`timeStamp`	RTP time stamp for the presentation of the sample data. This time stamp has already been adjusted by edits, edit rates, etc.
`duration`	Duration (in RTP time scale) of the sample.
`playOffset`	Offset within the media sample itself. This is only used for media formats where a single media sample can span across multiple time units. QuickTime Music is an example of this, where a single sample spans the entire track. For most video and audio formats, this will be 0.
`playRate`	1.0 (`0x00010000`) is normal. Higher numbers indicate faster play rates. Note that timeStamp is already adjusted by the rate. This field is generally of interest only to audio packetizers.
`flags`	`kRTPMPSyncSampleFlag` is set if the sample is a sync sample (key frame).
`sampleDescSeed`	If the sample description changes, this number will change.
`sampleDescription`	The sample description for the given media sample
`sampleRef`	Private field.
`dataLength`	Size of media data.
`data`	Pointer to media data.
`releaseProc`	If set, you need to call it when you are finished with the sample data.
`refCon`	Pass to release procedure.
`outFlags`	Set to `kRTPMPStillProcessingData` if you are not done with the sample. This will cause your idle routine to be called.

Note: The caller owns the `RTPMPSampleDataParams` struct. Your media packetizer must copy any fields of the struct it wants to keep. The caller will maintain only the media data in the struct until you call the release procedure.

If you can't do all the processing of the sample data in response to the `RTPMPSetSampleData` function, sets `outFlags` to `kRTPMPStillProcessingData` and do the work in your `RTPMPIdle` function:

```
// do work here if you need to -- give up time periodically
// if you're doing time consuming operations
pascal ComponentResult RTPMPIdle(RTPMediaPacketizer rtpm,
                SInt32 inFlags, SInt32 *outFlags);
```

This call is made periodically if you set `outFlags` to `kRTPMPStillProcessingData` in your `RTPMPSetSampleData` routine. If you need more time, sets `outFlags` to `kRTPMPStillProcessingData` in `RTPMPIdle` as well. This will cause `RTPMPIdle` to be called again. When you are finished packetizing the sample data you were passed in the last `RTPMPSetSampleData` call, sets `outFlags` to 0.

If you do work in the `RTPMPIdle` function, the idle function needs to call the release procedure when you are done with the sample data.

Calling the Packet Builder

The actual work your packetizer does in the `RTPMPSetSampleData` or `RTPMPIdle` function consists of creating packets. This is done by making calls to a packet builder component. The packet builder functions are defined in `QTStreamingComponents.h`.

The first thing you will need to do is to begin a new packet group. To start a new group of packets, call `RTPPBBeginPacketGroup`. Packets in a single group have the same RTP time stamp (and go into the same hint sample, if the data is being used to create a hint track).

Next, you will want to add one or more packets to the group.

- Call `RTPPBBeginPacket` to begin a single network packet.
- Call `RTPPBAddPacketLiteralData` to add literal data to the packet, such as header information.
- Call `RTPPBAddPacketSampleData` to add a sample reference to the packet, such as the sample data specified in `inSampleDataParams` in a call to your packetizer's `RTPMPSetSampleData` function.

You may want to packetize the same data repeatedly. There are utility functions to make this easier. Begin by passing in `RTPPacketRepeatedDataRef` to `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`, depending on whether you want to repeat literal data or sample data. The packet builder will store a copy of the data and return a data reference. You then call `RTPPBAddPacketRepeatedData` with the data reference whenever you want to add copies of the data. Call `RTPPBReleaseRepeatedData` when you are done with the data. The packet builder will maintain a copy of the data until you release it.

End a packet by calling `RTPPBEndPacket`. End the group of packets by calling `RTPPBEndPacketGroup`.

Note: It is legal to have more than one packet group open at a time, or to have more than one packet open at a time. Packet groups are sent in the order they are closed (`RTPPBEndPacketGroup`). Packets within a group are also sent in the order they are closed (`RTPPBEndPacket`).

Flush and Reset Routines

Your packetizer may be called with commands to flush its input buffer or reset for a new session at any time. This normally happens at the end of a sequence or when a transmission is interrupted.

The flush command is given when your packetizer needs to finish any pending work:

```
pascal ComponentResult RTPMPFlush(RTPMediaPacketizer rtpm,
                SInt32 inFlags, SInt32 *outFlags);
```

If your packetizer has not yet written all the data from the last `RTPMPSetSampleData` call to the selected packet builder, upon receiving the flush command it should write any pending data, flush its input buffer and set `outFlags` to 0. The `inFlags` value is currently always 0.

The reset command is given when your packetizer needs to stop packetizing, reset its state, and prepare for new orders:

```
pascal ComponentResult RTPMPReset(RTPMediaPacketizer rtpm,
                SInt32 inFlags);
```

The `inFlags` value is currently always 0.

Flush your input buffer. Do not send any buffered data, because in all probability there is no connection for you to send on. Reset your packetizer's state; it should be the same as if it had just been opened and initialized.

Packet Reassemblers

A packet reassembler extracts meaningful chunks of data, such as video frames, from streams of RTP packets. A reassembler can be specific to a particular media type and compression format, such as a reassembler for Sorenson video, or it can be more generalized, such as a reassembler for any uncompressed audio, or even a reassembler for any QuickTime media.

Streaming media over RTP generally involves some packet loss. It is the responsibility of the reassembler to perform any loss recovery that goes beyond discarding data chunks that contain lost packets.

Writing a Packet Reassembler

QuickTime includes a base reassembler that performs most of the routine work of packet reassembly. Your packet reassembler sets flags that control the behavior of the base reassembler. Your reassembler can also implement several functions to override the base reassembler, essentially taking over from it at almost any point.

The reassembler component must implement several functions, and must also provide a public component resource that describes the type of media, compression, and track characteristics that the reassembler supports. This resource also provides information on the reassembler's relative speed and the format's ability to handle loss.

The following functions can be implemented by packet reassembler components. As noted, some of these functions must be implemented in your component, some can be delegated to the base component, and some are base component utility functions that your component can call.

- `RTPRssmInitialize`
- `RTPRssmReset`
- `RTPRssmComputeChunkSize`
- `RTPRssmAdjustPacketParams`
- `RTPRssmCopyDataToChunk`
- `RTPRssmSendPacketList`
- `RTPRssmGetTimeScaleFromPacket`
- `RTPRssmSetInfo`

- RTPRssmGetInfo
- RTPRssmSetCapabilities
- RTPRssmGetCapabilities
- RTPRssmSetPayloadHeaderLength
- RTPRssmGetPayloadHeaderLength
- RTPRssmSetTimeScale
- RTPRssmGetTimeScale
- RTPRssmNewStreamHandler
- RTPRssmSendStreamHandlerChanged
- RTPRssmSetSampleDescription
- RTPRssmGetChunkAndIncrRefCount
- RTPRssmSendChunkAndDecrRefCount
- RTPRssmSendLostChunk
- RTPRssmClearCachedPackets
- RTPRssmReleasePacketList
- RTPRssmIncrChunkRefCount
- RTPRssmDecrChunkRefCount

Reassembler Component Type and Subtype

Packetizers have a component type of `kRTPReassemblerType` (`'rtpr'`). The subtype can be any four-character combination. Note, however, that all-lowercase types are reserved by Apple.

The 'rsmi' Public Resource

A reassembler must provide a public resource of type `'rsmi'`. The public resource contains information about the capabilities of a given reassembler. This information lists the RTP payload types the reassembler can work with. In addition, it provides information about the reassembler's performance characteristics, specifically its speed and ability to recover from lost packets.

If more than one reassembler is available for a given RTP payload type, QuickTime will choose the one with the best performance characteristics, such as speed or ability to deal with packet loss.

The format of the public resource is defined in `QTStreamingComponents.r` as follows:

```
type 'rsmi' {
    array infoArray {
        align long;
        longint = $$CountOf(characteristicArray); /* Array size */
        array characteristicArray {
            hex longinttag;
            hex longint value;
```

```
        };

hex longintpayloadFlags;
    /* kRTPPayloadTypeStaticFlag or kRTPPayloadTypeDynamicFlag */
        byte    payloadID;      /* if static payload */
        byte = 0;
        byte = 0;
        byte = 0;
        cstring;                    /* if dynamic payload */
    };

#define kRTPPayloadSpeedTag 'sped'/* 0-255, 255 is fastest */ ]

#define kRTPPayloadLossRecoveryTag 'loss'
    /* 0-255, 0 can't handle any loss, 128 can handle 50% packet loss */

#define kRTPPayloadTypeStaticFlag 0x00000001
#define kRTPPayloadTypeDynamicFlag 0x00000002
```

The payload flags field is set to `kRTPMPPayloadTypeDynamicFlag` if the reassembler handles a dynamic payload type, or `kRTPMPPayloadTypeStaticFlag` if it handles a static type.

The payload ID field of the `'rsmi'` resource is set to the IETF-defined RTP payload value if a static payload type is used.

The C string contains the RTP payload type text for dynamic types.

A declaration in a .r file might look like this:

```
resource kRTPReassemblerInfoResType (128) {
    {
        {
            kRTPPayloadSpeedTag, 128,
            kRTPPayloadLossRecoveryTag, 50
        },
        kRTPPayloadTypeDynamicFlag, 0, "x-oval"
    }
};
```

This resource indicates that the reassembler is of average speed and that it can handle 50% packet loss while still providing some meaningful data. It handles the dynamic RTP payload type identified by `x-oval`.

The speed tag is relative to other reassemblers of the same type. A value of 128 is a reasonable default.

The Base Reassembler

Your packet reassembler relies on a base reassembler component to do most of the routine reassembly work. Your reassembler can modify some of the base reassembler's default behaviors simply by setting flags. With a few exceptions, your reassembler implements reassembler functions only when it needs to override the normal behavior of the base reassembler.

The base reassembler's main function is to assemble incoming packets into "chunks", then pass the chunks to a stream handler. A chunk is the amount of data useful to a particular stream handler, such as a video frame or twenty milliseconds of audio.

Your reassembler must specify the type of stream handler needed, but your reassembler doesn't work with stream handlers directly. The base reassembler does this for you.

The default behavior of the base reassembler is as follows:

1. The base reassembler fills out an `RTPRssmPacket` struct for each packet it receives.
2. The base reassembler puts the packets into a packet list, which corresponds to a data chunk. By default, the packet list contains all the packets with the same RTP transmission time, which is also the sample time. The base reassembler will start a new packet list when it receives a packet with a different RTP transmission time or with the RTP marker bit set. Your reassembler can change this default by telling the base reassembler that every packet is a chunk.
3. Once a packet list is complete, the base reassembler creates a chunk from it. By default, the chunk is made by concatenating the payload of each packet in the packet list. By default, the payload is assumed to be the entire contents of the packet following the RTP header and the payload header. The base reassembler assumes a zero-length payload header (no payload header) unless your reassembler sets a nonzero payload header length as the default.
4. By default, the base reassembler discards chunks with missing packets. The base reassembler can be set to inform your reassembler that a chunk has missing packets.

By implementing certain functions, you can cause the base reassembler to call your reassembler at one of several points to override or modify the default behavior:

Implement this function	Your reassembler will be called
`RTPRssmAdjustPacketParams`	When each packet is received, after the base reassembler fills out the `RTPRssmPacket` struct
`RTPRssmSendPacketList`	When the packet list is complete
`RTPRssmComputeChunkSize`	When the chunk size is calculated
`RTPRssmCopyDataToChunk`	When the data is copied into the chunk record

This is discussed further in the sections "Handling Packets Yourself" (page 51) and "Handling Chunks Yourself" (page 53).

Opening Your Reassembler

QuickTime may open your packet reassembler to check its version or to get information. It may then close your reassembler without ever initializing it or using it to process packets.

Your reassembler component must be able to perform the standard component functions, such as `_Version`, and its specific `RTPRssmGetInfo` function, without being initialized.

When opened, your packet reassembler must open a base reassembler component. Your reassembler should delegate any functions it does not implement. It should also support the `_Target` call.

A typical reassembler's `_Open` and `_Target` functions might look like this:

```
pascal ComponentResult RTPOvalRssm_Open(RTPOvalRssmGlobalsPtr globals,
                                        ComponentInstance self)
{
#pragma unused (inGlobals)
    ComponentResult err;

    globals = (RTPOvalRssmGlobalsPtr)
                              NewPtrClear(sizeof(RTPOvalRssmGlobalsRecord));
    if ( (err = MemError()) != noErr )
        goto exit;

    SetComponentInstanceStorage(self, (Handle)globals);
    globals->self = self;
    err = OpenADefaultComponent(kRTPReassemblerType, kRTPBaseReassemblerType,
                                &globals->delegateComponent);
    if ( err == noErr )
        err = RTPOval_Target(globals, self);
exit:
    return err;
}

pascal ComponentResult RTPOval _Target(RTPOvalRssmGlobalsPtr inGlobals,
                                ComponentInstance inParentComponent)
{
    inGlobals->parent = inParentComponent;
    return ComponentSetTarget(inGlobals->delegateComponent, inParentComponent);
}
```

Initialization

Your reassembler must implement the `RTPRssmInitialize` function. When initialized, your reassembler is passed an `RTPRssmInitParams` struct containing three initialization parameters:

```
struct RTPRssmInitParams {
    UInt32      reserved;
    UInt8       payloadType;
    UInt8       pad[3];
    TimeBase    timeBase;
    TimeScale   controlTimeScale;
};
typedef struct RTPRssmInitParamsRTPRssmInitParams;
```

Term	Definition
`payloadType`	One-byte identifier for the payload type. It is the same number as sent in the RTP header. This is useful if your reassembler handles more than one format.
`timeBase`	The timebase for the presentation.
`controlTimescale`	The timescale used for actions such as start, stop, etc. This is independent of the timescale used for the media. For example, the control timescale could be 600 while the media timescale could be 90000.

During initialization, your reassembler should open a stream handler of the appropriate type. For example, if your reassembler works with H.261 packets, it should open a video stream handler. Stream handler types are specified in the same manner as track types (video is `videoMediaType`, and so on). There are currently stream handlers for audio, video, text, and MIDI.

Your reassembler opens a stream handler by calling `RTPRssmNewStreamHandler`. You must specify the stream handler type when you open it. If your reassembler handles multiple media types, it can open a stream handler later, after it learns what kind of media is in the stream.

You should initialize the sample description of the stream handler when you open it, if possible. It can be set or changed later if necessary. The stream handler will be unable to process any data until its sample description is set.

Your reassembler should also initialize the timescale of the stream handler at this time. If your reassembler needs to get the timescale from the stream, it can monitor incoming packets and set the timescale when it is known. (No chunks will be sent to the stream handler until its timescale is set).

During initialization, your reassembler should call `RTPRssmSetCapabilities` with any initial flags to control the base reassembler's default behaviors, such as `kRTPRssmEverySampleAChunkFlag` or `kRTPRssmTrackLostPacketsFlag`.

You should also call `SetPayloadHeaderLength` at this time if you know the payload header length for your packets.

A typical reassembler initialization function might look like this:

```
EXTERN_API( ComponentResult )
RTPRssmOVAL_Initialize(
RTPHOVALGlobalsPtrinGlobals,
RTPRssmInitParams *inInitParams )
{
#pragma unused(inInitParams)
ComponentResulterr = noErr;
ImageDescriptionHandle imageDesc;
SInt32     flags = 0;

inGlobals->fTimeScale = kOVALRTPTimeScale;

flags = kRTPRssmQueueAndUseMarkerBitFlag + kRTPRssmTrackLostPacketsFlag;
err = RTPRssmSetCapabilities(inGlobals->delegateComponent, flags, -1L );
    if (err == noErr) {
        imageDesc = __GetMyImageDesc( inGlobals );

        if( imageDesc )
        {
            err = RTPRssmNewStreamHandler(inGlobals->delegateComponent,
              VideoMediaType, ( SampleDescriptionHandle ) imageDesc,
              inGlobals->fTimeScale, NULL);
        }
        else
            err = memFullErr;
    }
    return err;
}
```

Setup and Information Functions

If you do not open a stream handler and set its timescale during initialization, you must implement the `RTPRssmGetTimeScaleFromPacket` function. If you have not set the stream handler's timescale, the base reassembler will call your reassembler's `GetTimeScaleFromPacket` function with every incoming packet until a stream handler is open and its time scale is set.

If you cannot determine the timescale based on the contents of the packet, return `qtsUnknownValueErr` or a 0 timescale.

```
EXTERN_API( ComponentResult ) RTPRssmGetTimeScaleFromPacket(
    RTPReassembler rtpr,
    QTSStreamBuffer *inStreamBuffer,
    TimeScale * outTimeScale);
```

The `RTPRssmGetInfo`, `RTPRssmSetInfo`, and `RTPRssmHasCharacteristic` functions can be called at any time, even prior to initialization. `RTPRssmHasCharacteristic` is called to determine what features your reassembler supports. `RTPRssmGetInfo` is used to get information from your reassembler. `RTPRssmSetInfo` could be used to send information to your reassembler, but there are currently no selectors that do this.

These functions are commonly used if your reassembler supports passing of non-media data, such as transformation matrices. The functions are typically called after your reassembler reports a change in a non-media parameter by calling `RTPRssmSendStreamHandlerChange`, as described in "Passing Non-Media Data" (page 31). The selectors used for passing non-media data are the same for `RTPRssmHasCharacteristic` and `RTPRssmGetInfo`. These selectors begin with `kQTSSource` and are defined in `QTStreaming.h`.

Delegate any selectors you do not support or do not understand to the base reassembler for all of these functions:

```
EXTERN_API( ComponentResult ) RTPRssmSetInfo(
    RTPReassembler    rtpr,
    OSType           inSelector,
    void *           ioParams) ;

EXTERN_API( ComponentResult ) RTPRssmGetInfo  (
    RTPReassembler rtpr,
    OSType     inSelector,
    void *     ioParams) ;

EXTERN_API( ComponentResult ) RTPRssmHasCharacteristic(
    RTPReassembler rtpr,
    OSType      inCharacteristic,
    Boolean *   outHasIt) ;
```

Handling Packets Yourself

Your packet reassembler can be called when each packet is received. If you have not yet opened a stream handler and set its timescale, you will receive a `RTPRssmGetTimeScaleFromPacket` call for each received packet.

Once the stream handler's timescale is set, the default behavior for the base reassembler is to fill out an RTPRssmPacket struct for each incoming packet, then to add each packet to the current packet list. The base reassembler starts a new packet list when a packet's RTP marker bit is set, or the RTP timestamp changes, or if your reassembler has set the kRTPRssmEveryPacketAChunkFlag in RTPRssmSetCapabilities.

If you want to fill out or modify the RTPRssmPacket struct for each packet yourself, because you use variable payload header lengths, for example, implement the RTPRssmAdjustPacketParams function.

```
EXTERN_API( ComponentResult ) RTPRssmAdjustPacketParams(
    RTPReassembler    rtpr,
    RTPRssmPacket     *inPacket,
    SInt32            inFlags) ;
```

The inPacket parameter points to the RTPRssmPacket struct for the current packet. This structure has already been filled in by the base reassembler:

```
struct RTPRssmPacket {
    struct RTPRssmPacket      *next;
    struct RTPRssmPacket      *prev;
    QTSStreamBuffer           *streamBuffer;
    Boolean                   paramsFilledIn;
    UInt8                     pad[1];
    UInt16                    sequenceNum;
    UInt32                    transportHeaderLength;
    UInt32                    payloadHeaderLength;
    UInt32                    dataLength;
    SHServerEditParameters    serverEditParams;
    TimeValue64               timeStamp;
    SInt32                    chunkFlags;
    SInt32                    packetFlags;
}; typedef struct RTPRssmPacketRTPRssmPacket;
```

Field	Definition
next	The next packet in the list; NULL if this is the last packet.
prev	The previous packet in the list; NULL if this is the first packet.
streamBuffer	The stream buffer containing the packet data.
pad	Not used.
sequenceNum	The sequence number associated with the packet. Sequence numbers are unsigned 16 bit numbers and wrap around. (65535 is followed by 0, 1, 2, etc). Sequence numbers start at arbitrary numbers.
transportHeader-Length	The length of the RTP header. The payload specific part of the packet begins immediately after the transport header. Do not assume that the RTP header length is 12.
payloadHeaderLength	The length of the payload header. The payload header (if any) starts immediately after the RTP header. This number gets filled in by the base reassembler by using the number passed into RTPRssmSetPayload-HeaderLength. It can also be modified by the derived reassembler.

Field	Definition
dataLength	The length of the payload data. This is usually the length of the packet minus transport header length minus payload header length. The base reassembler will set the default value to that. Your reassembler can modify the number.
timeStamp	The 64 bit timestamp associated with the packet in the stream timebase (not the movie timebase). The timestamp sent in the RTP header is 32 bits and wraps around. The 64 bit timestamp in this structure accounts for the wraparound. The lower 32 bits are exactly the timestamp sent in the RTP header. The upper 32 bits are the number of wraparounds.
chunkFlags	Flags that should be set in the chunk that is sent to the stream handler. The base reassembler calculates the value that is set in the SHCHunkRecord structure by OR-ing all the chunk flags in the RTPRssmPacket list. Your reassembler should fill in these flags as appropriate. Flags you could set include kSHChunkFlagSyncSample (set this flag if the chunk is a sync sample) or kSHChunkFlagDataLoss (set this flag if the chunk has data loss in it; for example, if there was data loss in the video frame and the reassembler did partial recovery).
packetFlags	The base reassembler fills in this value. Defined flags include kRTPRssmPacketHasMarkerBitSet = 0x00000001 (the base reassembler fills in this value from the RTP header) and kRTPRssmPacketHas-ServerEditFlag = 0x00010000 (the derived reassembler should set this flag to 1 if this packet has a server edit; you should also fill in the serverEditParams field).

Important: Do not modify the data in the stream buffer. For example, if you have to flip bytes for endian differences between network byte order and the native byte order, copy the resulting data. Do not flip the bytes in place. Other components might have references to the same data.

Handling Chunks Yourself

The base reassembler automatically builds chunks from packets and sends them to the stream handler you have opened. If any packets are missing from the chunk, the base reassembler discards the entire chunk.

You can take over from the base reassembler at several points in the chunk-building process by implementing the appropriate function. Do this if your reassembler performs loss recovery, or if the base reassembler's default behavior needs to be modified to correctly build chunks for the stream handler.

Your reassembler is responsible for creating and sending the chunk, beginning at whatever point you take over, and continuing until the chunk is sent or discarded.

The process of creating a chunk begins when the base reassembler has a complete packet list. To take over at this point, implement RTPRssmSendPacketList. You might want to do this if you were altering the packet list for loss recovery.

If you implement `RTPRssmSendPacketList`, you are responsible for deleting the packet list when you are through with it, by calling `RTPRssmReleasePacketList`.

Next, the base reassembler calculates the chunk size. It does this by summing the payload size for each packet in the list. The payload size is calculated by subtracting the RTP header and the payload header from the packet size. To take over at this point, implement `RTPRssmComputeChunkSize`. You might want to do this if you need to add data to the chunk that isn't in the packets, or to use a different method for calculating the payload size.

If you implement `RTPRssmComputeChunkSize`, you will need to allocate a chunk of the appropriate size. To do this, call `RTPRssmGetChunkAndIncrRefCount`. This will allocate the chunk and set the number of references to it to 1. QuickTime will maintain the chunk until the number of references to it is 0. The reference count is automatically decremented when you send the chunk. You can increment the counter by calling `RTPRssmIncrChunkRefCount` if you want QuickTime to preserve the chunk for later use (by substituting for a lost chunk, for example).

Once the chunk size has been calculated and the chunk has been allocated, the base reassembler moves the data from the packets into the chunk. The data moved will be a simple concatenation of the packet payloads. The payload size and offset within each packet are calculated in the same manner as used for calculating chunk size. To be able to take over at this point, implement `RTPRssmCopyDataToChunk`. You would need to do this to modify the bytes at packet boundaries for an H.261 packet reassembler, for example.

Bear in mind that if you implement `RTPRssmSendPacketList`, you are responsible for the steps performed in `RTPRssmComputeChunkSize` and `RTPRssmCopyDataToChunk`. Similarly, if you implement `RTPRssmComputeChunkSize`, you must perform the steps for `RTPRssmCopyDataToChunk`. Only the first of these functions that you implement will be called. You take over the chunk-building process from there.

If you implement any of the three functions just discussed, you are responsible for sending the chunk by calling `RTPRssmSendChunkAndDecrCount`.

Other things your reassembler might do at this point are to

- Tell the stream handler that the sample description has changed (`RTPRssmSetSampleDescription`).
- Tell the stream handler that some non-media data has changed, such as a transformation matrix or sound volume (`RTPRssmSendStreamHandlerChanged`). You may receive a series of `GetInfo` calls to determine what has changed.
- Tell the stream handler not to send the chunk. The reference counter will be decremented; if it becomes zero, the chunk is deallocated (`RTPRssmSendLostChunk`).
- Tell the base reassembler to keep a copy of this chunk for your later use (`RTPRssmIncrChunkRefCount`). Be sure to increment the ref count before you send the chunk. You must eventually call `RTPRssmDecrChunkRefCount` for every call to `RTPRssmIncrChunkRefCount` or you will create a memory leak.

Reset and Clear Cache Functions

Your reassembler can be called at any time with the `RTPRssmReset` function, which you must implement. Reset all your variables, release any chunks being held for you, and prepare for a new run of data. At the end of an `RTPRssmReset` function, your state should be identical to its state after the first `RTPRssmInitialize` call.

You can instruct the base reassembler to release any packets in the packet list it is currently building by calling `RTPRssmClearCachedPackets`. You might do this if you are handling packets yourself and you determine that the list the base reassembler is building should be discarded.

Document Revision History

This table describes the changes to *QuickTime Streaming Guide*.

Date	Notes
2006-01-10	Removed obsolete material and changed title from "QuickTime Streaming."
2002-09-17	New document that describes the QuickTime streaming API.